Geo-Replicating Federated File Systems

Benjamin Bengfort and Pete Keleher
Department of Computer Science
University of Maryland, College Park, MD, USA
{bengfort,keleher}@cs.umd.edu

***Abstract*—Groups of strongly consistent devices can efficiently order events under ideal (data center) conditions, but become less effective in dynamic and heterogeneous environments. Eventually consistent devices efficiently tolerate both faults and dynamic conditions but are slow to converge on a single ordering of system events.**

We propose "federated consistency", which combines the strengths of both approaches into a single protocol, and show how to map a distributed file system onto a federated system. Federated groups use a strongly consistent inner core of devices to maintain a totally ordered, fault-tolerant sequence of events. A cloud of eventually-consistent devices disseminates orderings and enables progress despite varying connectivity and partitions. Though the constituent sub-protocols take different (nearly opposite) approaches to resolving conflicts; we show that use of a forte number allows them to inter-operate effectively. We use a discrete event simulation to show that a group of federated devices can obtain the key advantages of both approaches.

I. Introduction

The rise of on-demand computing resources and the Cloud has made distributed systems the default approach to scaling applications for many users in a variety of geographic locations. In particular, data replication is used to increase availability, throughput, durability, and fault tolerance by ensuring that objects can be accessed on multiple servers as locally as possible. Although some coordination is necessary to ensure that replication happens correctly, many systems favor a relaxation in consistency in order to meet the performance requirements of modern, mobile applications. This is partially because the application layer can define its own mechanisms for handling differently consistent behavior but it is primarily because such systems are implemented in data center contexts that enjoy stable, low-latency connections which allow optimistic approaches and provide consistent views of data *most of the time* [8], [9].

We might, therefore, generally categorize most modern distributed storage systems and NoSQL databases as not having strong consistency and using some consensus algorithm for control and synchronization when necessary. As a result consistency is usually described in a discrete, data-centric fashion: weak or strong; eventual, causal, or sequential and no longer described in client-centric terms [10]. As replication becomes more prevalent, however, it is not enough to simply lay the burden of conflict at the feet of clients and there has been recent interest in instead redefining consistency along a spectrum whose dimensions are the strictness of ordering writes and the potential staleness of reads [45], [31], [1], [4], [26].

By defining consistency in terms of ordering and staleness it is easy to see that the root cause of the tradeoff between performance and correctness is message latency, where a common case is when messages do not or cannot arrive due to node failure or network partitions. It has been noted that message latency is the key factor in determining "how consistent" a system is either due to staleness in eventually consistent systems [7] or by preventing progress in sequential consistency systems implemented with consensus [20]. The advent of distributed storage as a service has allowed systems to adapt consistency at runtime by taking advantage of a stable network environment [12], [13], [25], [37], [11] and has shifted the focus away from replication in weakly-connected, dynamic, or mobile networks. We believe that local, user-oriented distributed systems should augment cloud services rather than be replaced by them; and in some cases, such as disaster recovery or search and rescue, may be the only available system.

In this paper we present a novel approach to flexible consistency via the federation of a heterogeneous system of replica servers that implement a variety of consistency protocols in response to local policies and requirements. As a result, individual replicas in the system can respond and adapt to a changing network environment while providing as strong a local guarantee or minimum quality of service as required. The global state of a federated system is defined by the replica topology and their interactions, such that if a subset of nodes implement stronger consistency models, then the global probability of conflict is reduced. Conversely, a subset of nodes implementing a weaker consistency can increase global throughput. Indeed, we find that it is more often the tension between local vs global views of consistency that cause greatest concern in terms of application performance. Because each node can select and change local consistency policies, client applications local to the replica server have greater control of tuning consistency in response to mobile or dynamic network behavior, maximizing timeliness or correctness as needed.

We show that a federated consistency protocol can find a middle ground in the trade-off between performance and consistency, particularly between an eventually consistent system implemented via gossip-based anti-entropy [23] and a sequential consistency model implemented by the Raft quorum consensus protocol [35]. By exploring these two extremes in the consistency spectrum we show that the overall number of inconsistencies in the system is reduced from the homoge-

neous eventual system and that the access latency is decreased from the homogeneous sequential system. Moreover, because the global consistency of the system is topology-dependent, it can be said to have flexible or dynamic consistency. We have found that large systems with variable latency in different geographic regions can perform well by allowing most nodes to operate in an optimistic fashion, but also maintaining a strong central quorum to reduce the amount of global conflict.

The rest of the paper is organized as follows: Section II describes background and defines terms. Section III describes implementation of eventually consistent and Raft clouds, focusing on the non-standard aspects of our implementations, which focus on supporting file systems in wide-area and mobile environments. Section IV describes our Federated protocol with eventual and sequentially consistent sub-protocols. Section V explores simulated performance of the three protocols, and Section VI concludes.

II. Background and Definitions

Our targeted environment is wide-area: we assume replicas are distributed across multiple geographical regions, where communication within a region is fast and cheap and inter-region communication is expensive. We define consistency in terms of the ordering of operations that change the state of a replica. In the context of a wide-area file system, those operations could be individual `write()` systems calls, though this would be inefficient. Most wide-area file systems aggregate individual accesses through *Close-To-Open* (CTO) consistency, where file reads and writes are "whole file" [21], [24], [34]. A file read ("open") is guaranteed to see data written by the latest write ("close"). This approach satisfies two of the major tenets of session consistency: `read-your-writes` and `monotonic-writes`, but not `writes-follow-reads` [10], [41], [44].

The system operations to be replicated are therefore a series of file writes. Each file has meta-information that includes a unique name and a monotonically increasing version number. A file write includes the file name, the parent version of the file to which the write is being applied, versions and names of any other dependencies, the replica ID where the write occurred, and an array of *blob* (opaque chunks of data) IDs that encompass the file data at the conclusion of the write. A file read simply looks up the latest local version of that file.

Our file system, like many modern file systems, decouples metadata *recipes* [43], [15], [40], [39], [36] from file data storage. Metadata includes an ordered list of *blobs*, which are opaque binary chunks. When a file is closed after editing, the data associated with the file is *chunked* into a series of variable-length blobs [34], identified by a hashing function applied to the data [38]. Since blobs are effectively immutable [18], or tamper-evident, (blobs are named by hashes of their contents), we assert that consistent metadata replication can be decoupled from blob replication. Accesses to file system metadata becomes the operations or entries in replicated logs. Metadata is therefore replicated through the system, allowing any file system client to have a complete view of the file system namespace, even while not caching any file data.

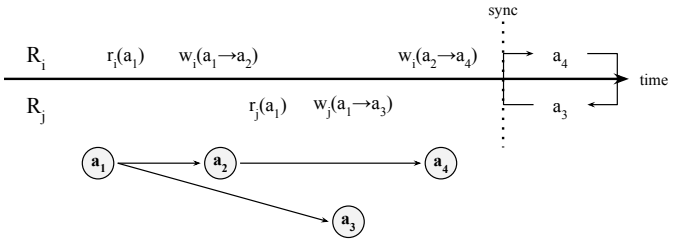

Fig. 1. Accesses before synchronization cause stale reads and forked writes.

A. Operation Logs

Meta-data writes are replicated through per-replica *logs*. Consistency is therefore expressed in terms of an ordered operation log, replicated across all system replicas, where operations consist solely of file writes. If all logs are identical, the same file writes are applied in the same order at each replica, and the file system will appear the same on all replicas. As such, we use a data-centric model of correctness that concentrates primarily on two metrics: log operation ordering and read staleness [10]:

1) *Ordering* refers to how closely individual logs adhere to the abstract global ordering. A strict ordering requires every single log to be exactly the same, whereas weaker ordering allows some divergence in the order writes are stored in the log.
2) *Staleness* refers to how far local logs are behind the latest version of the log globally, and can be either expressed by the average latency of replicating new versions, or simply by how far behind the latest is the average replica.

Most data-centric consistency models do not consider staleness, but instead refer to guarantees on ordering strictness and the method by which updates are applied to the state of the replica. However, the goals of strict ordering and minimal staleness can be in conflict. For example, a strict ordering criteria may require a new object version to be delayed until dependencies are satisfied, causing reads to become more stale.

B. Forks

In addition to tracking staleness directly, we also use as a metric the primary symptom of stale reads and writes, forks:

> A *fork* occurs when two replicas concurrently write a new version to the same parent object.

Forks introduce inconsistency because they allow multiple potential orderings of operation logs. Forks are primarily a symptom of staleness; e.g. the second writer wrote to a stale version of the object.

Almost all consistency models, with the exception of *linearizability* [19], are susceptible to forks because they allow stale reads to occur. Object *coherence* requires an object's version history to be a linear sequence. Object forks violate

coherence and occur, for example, when replicas i and j read object version a_1 and then concurrently attempt to write new versions: $W_i(a_1 \rightarrow a_2)$ and $W_j(a_1 \rightarrow a_3)$ (Figure 1). A delay in replicas R_i and R_j synchronizing could lead to one of them continuing further down one of the fork paths, e.g. $W_i(a_2 \rightarrow a_4)$. Forks can be caused by concurrent reads, but the fork between a_2 and a_3 actually occurs because R_j's read is stale.

C. Consistency

A broader discussion of consistency levels should include at least eventual consistency [42], causal consistency [3] (the highest level allowing high availability [6]), sequential consistency [29], and linearizability [19]. For reasons of space, however, we limit this discussion to the models most useful for our purposes: eventual and sequential consistency.

Eventually consistent logs, no matter their ordering, must eventually have identical final writes for each object in the namespace (recall that writes are whole-file with CTO consistency). This suggests that eventual consistency requires some *anti-entropy* mechanism to propagate writes and a policy to handle convergence [42]. Eventual consistency is very popular for NoSQL databases and hosted distributed storage services [14], [28] because it allows optimistic implementations. Conflict resolution is often left to the application layer, but in practice most applications can handle some inconsistency. The low latencies in cloud data centers often make windows of inconsistency rare and short-lived enough to be tolerable [8].

Eventual consistency convergence implemented by a *last writer wins* policy simply accepts all writes so long as they are more recent than the latest local versions. Reads and writes are always performed locally, and therefore with little performance overhead. Eventually consistent logs may temporarily diverge as long as the final versions of each object eventually converge. As a result, the current version of an object on a single replica may alternate between writes to competing forks (a fairly weak semantic); it is up to the application to detect and compensate for the inconsistency.

Sequentially consistent logs must be ordered identically, though the logs of lagging replicas may only be prefixes of the latest log in the system [5]. Sequential consistency does not make guarantees about staleness (or the ordering of reads) but does require that all writes become visible in the same order [10]. Sequentially consistency can be implemented with consensus algorithms such as Paxos [30] or Raft [35] that coordinate logs by defining a transitive, global ordering for all conflicts. Alternatively, sequential consistency can be implemented with warranties – time-based assertions about groups of objects that must be met on all replicas before the assertions expire [33].

Stale reads are possible because of lagging replicas. However, only a single branch of a forked write can be committed to any copy of the log. Preventing forks would require either a locking mechanism or an optimistic approach that allowed operations to occur but rejects all but one branch (the approach discussed in our Raft implementation below). Write rejection requires the application to deal with dropped writes by either retrying or resolving conflicts and writing a new version.

III. Replication

A federated consistency model allows individual replicas to engage in replication according to locally-specified consistency policies. Each replica maintains its own local state, modified in response to local accesses and receipt of messages from remote replicas. Each replica sends messages to other replicas in order to propagate new writes. Therefore every replica can be seen as an event handler that responds to local access events, as well as remote messages, and generates more events (sent messages) in return. So long as every federated replica has an event handler for all types of RPC messages, federation only has to be defined at the *consistency boundaries*, that is when replicas of one consistency type send messages to that of another.

A. Gossip-Based Anti-Entropy

Eventually consistent replicas read and write locally without remote communication delays. Writes are propagated among replicas through periodic *anti-entropy* sessions that converge replicas towards the same state (e.g. reducing entropy, the divergence between the states of individual replicas) [23]. At routine intervals specified by the `anti-entropy delay` timing parameter, a replica will randomly select one of the other replicas in the system and send a `Gossip` message that logically contains the latest version of all objects in the replica's local log. On receipt of a `Gossip` message, a remote replica will compare the RPC object versions with those in its local log. If the RPC versions are later, it will append the later versions of the object to the log (*last-writer wins*). Any object versions later on the remote replica are returned to the originating replica in a `GossipResponse` message. As a result, our anti-entropy implementation is *bilateral*.

Forks are caused by staleness due to propagation delays. This *visibility* latency can be modeled as:

$$t_{visibility} \approx \frac{T}{4} \log_3 N + \epsilon \tag{1}$$

$\frac{T}{4}$ is the `anti-entropy delay` as computed from the network environment via a *tick* parameter, T (discussed below), and N is the number of replicas in the system. The epsilon parameter specifies the amount of added latency injected by imperfect gossip neighbor selections; $\epsilon = 0$ would mean that on every anti-entropy session each node perfectly selected another replica that had not seen the write being propagated.

B. Raft Quorum Consensus

We implement sequential consistency by replicating the operation log through the Raft consensus algorithm [35]. As background, each Raft replica must always be in one of three states: `follower`, `candidate`, or `leader`. All replicas start in the `follower` state. Raft is governed by two primary timing parameters: the `heartbeat interval`, which specifies how often the leader sends `AppendEntries`

messages that disseminate new operations and double as keep-alive messages, and the `election timeout`, an interval after which a follower may conclude that the leader is dead and attempt to become leader itself. Replicas will vote for a candidate if and only if the prospective leader's term is greater than their own and if they have not voted for any competing candidate. A candidate receiving a majority of votes becomes the new leader.

The Raft leader has the primary responsibility of serializing and committing new operations to the replicated log. To that end, the leader will broadcast periodic `AppendEntries` messages to all other Raft followers in order to maintain their leadership for the given term. A write access that originates at a follower is sent as a `RemoteWrite` to the leader, and the leader accepts writes in the order that they are received.

Because all writes originating at followers are forwarded to the leader, the leader can guarantee a sequential ordering of updates. Therefore on receipt of an `AppendEntries` message, followers simply add the entries to their log and respond with their last index. If a majority of followers append entries to their logs, the leader will mark those entries as committed and inform the followers the write has been committed on the next `AppendEntries`.

Our implementation differs from a generic implementation of Raft in that a leader that detects a fork – a write having a parent version that is already listed as a parent version in the log – rejects (drops) the later write. Also, our implementation reduces message traffic by sending `AppendEntries` messages only periodically, attempting to aggregate multiple writes into a single message.

Although all writes are sequentially ordered, the aggregation delay raises the possibility of several distinct read policies, including:

1) `read_committed` - Raft replicas only read the latest committed version of an object, which occurs at best on the *second* `AppendEntries` message after the `RemoteWrite` is sent to the leader. Committed writes are guaranteed not to be rolled back, but introduces significant delay and the possibility of staleness and version forks.
2) `read_latest` - Replicas read the latest version of the object in their log, even if it has yet to be committed. Moreover, replicas will read their own local writes (`read-your-writes`) rather than waiting for an `AppendEntries` to return their write. Reads are fast, but may return values that are never committed.
3) `read_remote` - Rather than read locally, simply request the latest version from the leader. This introduces the potential for additional latency, but may be faster if the expected message latency is less than the heartbeat interval.

Each of these options has critical implications for the likelihood of stale reads and writes in the system. Replicas would choose `read_committed` if the network was highly partition prone and messages from the leader were unstable and prone to being rolled back. `Read_remote` serves replicas well when the average message latency is far lower than the heartbeat interval, though this could be improved by making the heartbeat interval similar to the network latency. Intuition suggested and experimentation confirmed that `read_latest` is the most appropriate approach for a file system in our environment.

C. Timing Parameters

Both the anti-entropy and Raft protocols are parameterized by timing constraints that govern replication and therefore overall consistency. In order to select an `anti-entropy delay`, `heartbeat interval`, and `election timeout`, we must find some method of scaling the timing to the expected base latency of our system. We use a "tick" parameter, T, which is a function of the observed one-way message latency in the system specified as a normal distribution of latency and described by its mean, λ_μ and standard deviation, λ_σ. T is used to derive all timing parameters in the federated system. We use a conservative formulation that is big enough to withstand most variability:

$$T = 6(\lambda_\mu + 4\lambda_\sigma) \tag{2}$$

Ongaro and Ousterhout [35] use a more conservative parameter of $10\lambda_\mu$, which results in replication happening much more slowly than access events and causes large numbers of conflicts. Howard et. al [20] use an optimistic T parameter, $2\,(\lambda_\mu + 2\lambda_\sigma)$, which is too small to capture the variability in our target environments and leads to out-of-order `AppendEntries` messages in Raft, which can degrade performance.

Timing parameters are then defined in terms of T. For example, in order to ensure that eventual and sequential replicas send approximately the same number of messages (e.g. fixing the message budget in resource constrained environments) the timing parameters are as follows:

- `anti-entropy delay` $= \frac{T}{4}$
- `Raft heartbeat interval` $= \frac{T}{2}$
- `Raft election timeout` $= U(T, 2T)$

IV. Federated Consistency

A federated model of consistency creates heterogenous clouds of replicas that participate in different replication protocols. Global consistency and availability of the system is tuned by specifying different allocations of replicas of each type. Allocating all of one replication protocol, e.g. a homogenous eventual or Raft cloud, should behave equivalently to a homogenous system that does not implement federation. Therefore a key requirement of federated consistency is the integration of protocols with no performance cost to replicas participating at different, local consistency levels.

We expect that a federated model will allow an eventual cloud to benefit from lower data staleness and fork frequency by being connected to a strongly consistent, central [PJK: not true, update] consensus group. Similarly, Raft replicas should be able to use anti-entropy mechanisms to replicate data and continue writing even if the leader is unavailable and

no consensus can be reached to elect a leader. We integrate the systems by relying on the eventual replicas to disseminate orderings and cope with failures, but relying on the Raft replicas to choose the final operation ordering. In order to achieve this with no performance cost we must ensure that replicas can inter-operate both in terms of communication (message traffic) and consistency.

A. Communication Integration

All replication protocols are defined by their RPC messages and expected responses. On one level it is a simple matter to integrate the communication across protocols by ensuring that all replicas respond to all RPC message types and that those types are clearly defined. Integration occurs when a subset of replicas implements *more than one* replication protocol, or when rules are established for cross-communication to take advantage of the unique characteristics of a protocol or topology.

We integrate communication at Raft replicas by allowing them to participate in anti-entropy with the eventual cloud (but not with other Raft replicas). Because the Raft replicas are generally a small subset of the overall system, this type of integration ensures that the number of messages in the system does not scale according to the number of replication protocols being federated. Eventual replicas therefore "synchronize" with Raft replicas by exchanging `Gossip` RPC messages initiated from either the Raft or the eventual replica. If an eventual replica receives an `AppendEntries` or `VoteRequest` RPC it must respond with an RPC failure that indicates the quorum has changed, which leads to a joint consensus decision among the remaining Raft nodes.

Communication integration can also take advantage of the geographic topology of the system to localize non-broadcast forms of communication. Specifically, eventual replicas can prioritize their communication with Raft replicas or local replicas by modifying the random selection of pairwise anti-entropy. Eventual replicas select a neighbor by first deciding between synchronization with Raft or another eventual replica with probability P_{sync}. If synchronization is selected, then anti-entropy occurs with the *geographically nearest*, available Raft replica. If an eventual replica is selected then a second decision is made between selecting a neighbor in the local area or in the wide area with a probability P_{local}.

In the homogenous eventual case, only P_{local} has an effect on replication (homogenous Raft is broadcast oriented and therefore does not depend on these probabilities). By slightly favoring synchronization and local communication for anti-entropy, the system becomes more reliant on the Raft core group and therefore has stronger global consistency (fewer forks overall). Alternatively, lowering the likelihood of synchronization will allow the system to become less reliant on Raft, particularly when wide area outages are likely.

Varying communication between protocols in this way introduces an important question: does a Raft consensus group improve the global consistency because it broadcasts across the wide area or because it implements a stronger consistency model? We investigated the relationship between broadcast replication by implementing a special eventually consistent replica called the "stentor" replica. Stentor replicas conduct two anti-entropy sessions per replication interval, one across the wide area and one locally. We compared a federation of Raft and eventual with a federation of eventual and stentor and found that while stentor performs slightly better than homogenous bilateral anti-entropy, the Raft consistency model has a strong effect on how inconsistencies are handled.

B. Consistency Integration

Consistency integration occurs when communication occurs between replicas with different local consistency policies. When an Eventual replica receives a `Gossip` message from a Raft replica it accepts the most recent version, however the reciprocal Raft operation applies consistency policies such as rejecting forks. If a fork is detected at a Raft follower, it can drop the inconsistency without the leader, which allows the Raft consensus group to be more available. In order to improve performance, all Raft replicas keep local caches of forked or dropped writes to minimize the propagation of forks and prevent duplication of remote accesses to the leader. Even though a Raft follower notes a fork and does not propagate it, it is possible for the fork to arrive at another Raft follower that has yet to see it via anti-entropy propagation. In this case increasing P_{local} can prevent the eventual cloud from propagating forks around Raft.

However, straightforward integration of the eventual and Raft clouds actually performs worse than either cloud in isolation. The problem is that eventual and Raft replicas resolve fork conflicts in exactly opposite ways. Eventual replicas choose the last of a set of conflicting writes through a latest-writer wins policy, whereas Raft replicas effectively choose the first by dropping any write that conflicts with previously seen writes. Given conflicting writes w_i with timestamp t and w_j with timestamp $t+1$, the eventual replicas will converge to w_j because its timestamp is later; whether or not the timestamps reflect real time is not relevant. However, the Raft nodes will converge to whichever write first reaches the leader; there is no mechanism by which to override a write that has already been committed. The end result is that the eventual replicas might all converge on w_i and the Raft replicas on w_j, with neither write ever being replicated across the entire system. This disconnect arises from a fundamental mismatch in the protocols' approaches to conflict resolution. We could modify one or the other, but would then have a protocol that would not perform as well in a non-federated environment.

We resolve this issue by noting that if the strong central quorum can make a write accepted by the Raft replicas "more recent" than any conflicting write, even one with a later timestamp, then the eventual replicas will converge to the write chosen by the Raft replicas. We therefore extend each version number with an additional monotonically increasing counter called the *forte* (strong) version, which can only be incremented by the leader of the Raft quorum. Because the Raft leader drops forks, or any version that was not more

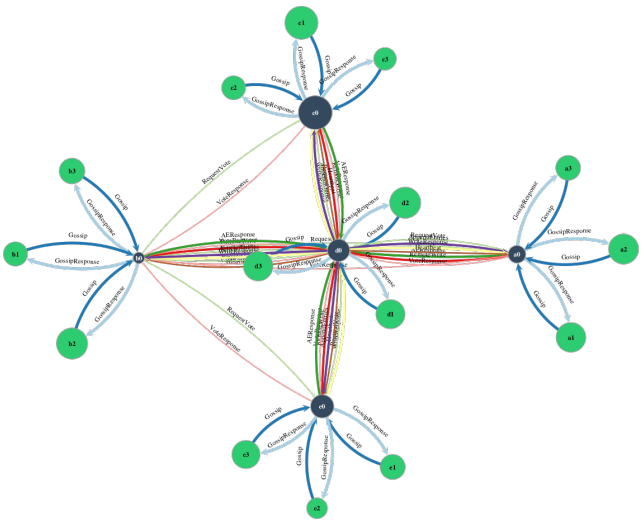

Fig. 2. Communication volume (line thickness) between heterogeneous replicas in a Federated topology.

recent than the latest commit version, incrementing the forte number on commit ensures that only consistent versions have their forte numbers incremented. The system then compares version recency is determined by comparing the forte number first and then the timestamp, allowing Raft to "bump" its chosen version to a later timestamp than any conflicting writes. This bump in forte number must be propagated to derived writes as well, as otherwise the increment of a write's forte number results in a timestamp later than that of writes derived it. On receipt of a version with a higher forte than the local, eventual replicas search for the forte entry in their local log, find all children of the update, and set the child version's forte equal to that of the parent.

V. Simulation Results

A. System Model

We investigate the effect of variable latency and the network environment on consistency by constructing a fully connected topology of replicas distributed among several geographic regions, as shown in Figure 2. Within each region, replicas enjoy stable, low-latency connections with their neighbors. However, across regions the latency is higher and the connections variable, meaning that out of order messages are more common across the wide area than in the local area.

In this type of topology we experience both replica failures, where a single replica stops responding to messages, and network partitions, where messages can not be exchanged with a single geographic region. In both cases, accesses may continue within a partitioned replica or subnet though they are not immediately replicated in the system. Partitioned replicas may fall behind the global state, and must be re-integrated into the network when the network interruption ceases.

We evaluate a *Federated* model of replication and consistency by creating a discrete event network simulation that allows us to flexibly configure several parameters. The simulation input has two parts: a topology that specifies the replicas and network environment, and a workload of access events to uniquely named objects (files). The simulation instantiates each replica as a process that executes read and write accesses to objects, generates replication messages, and handles messages from other replicas. Topologies specify each device as an independent replica by uniquely identifying it with device-specific configurations. By far the most important configuration option is a replica's *consistency* (or replication protocol), which determines the replica's behavior:

- *eventual*: Eventual replicas perform replication via anti-entropy by random selection of a neighbor to do pairwise gossip with. Eventual replicas can prioritize local vs. wide area replicas by specifying P_{local} – the likelihood of local neighbor selection.
- *Raft*: Raft replicas implement the Raft consensus protocol, electing a leader and forwarding writes to the leader to maintain a sequential ordering. If a write is forked, the leader will drop it in order to maintain our consistency guarantee.

The topology further specifies the *location* of each device, the *connections* between devices, and the *distribution* of message latency on a per-connection basis. Topologies were parameterized with two primary latencies specified as normal distributions (λ_μ, λ_σ): the *local area* latency, usually with a lower λ_μ and λ_σ than the *wide area* latency – e.g. the latency between devices in different locations. In order to compute the tick parameter, T, and specify the average latency in the simulation, latencies are given as worst case, wide-area latencies. Each topology could also set simulation-specific device-specific configurations, though we do not use that ability here.

The workload was specified as access trace files – time ordered access events (reads and writes) between a specific device and a specific object name. Each trace was constructed via a random workload generator, where a collection of available devices was specified along with a normal distribution of the delay between accesses (A_μ, A_σ), the number of objects, o, in the system, and a probability of conflict, P_c. To generate the workload, object names were assigned to each replica as follows: in a round-robin fashion, an object name was selected and assigned to a replica with probability P_c until each replica was accessing o objects. If $P_c = 1.0$ then every single replica would be accessing the same batch of objects, where as if $P_c = 0.0$ then each replica would access their own unique set of objects. From there, the A_μ, A_σ was used to generate accesses to objects in sequence, by selecting an object and reading and writing to it over time until some probability of switching objects occurred. In effect, the final workload simulates multiple replicas reading and writing at a moderate

pace for approximately one hour.

One example topology can be seen in Figure 2. Vertice size represents the number of accesses that occur at that location, and color represents the replica type. The edges are colored by RPC type and sized by the number of messages. Replicas have a preference for anti-entropy in the local area cluster, primarily synchronizing across the wide area via Raft replicas.

B. Experiments and Metrics

We conducted two primary experiments to test the behavior of a Federated consistency system against homogeneous Raft and Eventual systems. The first investigates behavior in the face of increasing failure rates, P_f and the second explores the effect of the network environment in terms of the mean latency, $\lambda\mu$ of the wide area. Our simulated topology consists of twenty (20) replicas distributed across five (5) geographical regions. Eventual replicas prefer other replicas within the same geographical region for anti-entropy. Our Federated topology consists of an inner core of Raft replicas distributed one per region, each co-located with several eventual replicas. Our experiments were driven by synthetic access traces containing approximately 29,000 accesses (depending on the experiment), with an average ratio of 54% of the accesses being reads to model the ratio of file reads to writebacks.

Our primary metrics are *stale reads*, *forked writes*, which can produce application-visible effects. We define forked writes as the number of writes that had more than one child (multiple writes had the same parent version), whereas reads are stale if they return anything other than the globally latest version. We also look at *write visibility*. Recall that a write is *visible* if and when it becomes replicated on all replicas. Any writes that do not become fully visible (stomped on through the eventually consistent policy of latest writer wins) are ignored. This metric is closely related to the *percent visible* metric — the average number of replicas a write is propagated to. Finally, we show one graph comparing write latencies. Read latencies are not meaningful as all read requests are satisfied locally by all of our protocols, and so have essentially zero latency.

C. Failure Rates

Figure 3 shows the portion of system reads that return stale data as the probability of outages, $P_f \in [0.0, 1.0]$, increases with a step of 0.1. The Eventual system deals with increasingly poor network conditions the best, as randomized anti-entropy partner selections allow writes to propagate through multiple paths. Eventually consistent systems are wide used precisely because of their ability to remain highly-available despite network failures and partitions [6], [7], [8]. The Federated system is able to leverage the Eventual subset of its replicas to route around failures almost as efficiently as the homogeneous Eventual system.

The multiple-paths ability also allows the Federated system to propagate writes quickly, as shown in Figure 3. In fact, Federated actually outperforms the Eventual system, possibly because the Raft quorum is able to quickly disseminate chosen writes during those periods when wide-area links are available.

D. Latency Variation

In the second experiment we investigate the effect of variable network latency on consistency protocols, as well as how the selection of the tick parameter model affects consistency for each system. In each of these environments we again evaluated three replication models: *Eventual*, *Raft*, and *Federated*.

Each simulation was then parameterized by a T parameter, computed by the wide area λ_μ and λ_σ. However the access mean, A_μ was fixed at approximately one access per replica every 3000ms. In effect, this meant that for approximately half the simulations (with the higher latencies), it was impossible for a write to become visible on another replica before a fork. Even though we fixed the conflict probability as $P_c = 0.5$ there was still enough conflict in the simulation to force each protocol to handle many forks.

Figures 5 and 6 show that write propagation is much faster and more effective in Raft than in Eventual, especially as network conditions deteriorate. For mean replication *latency* (visibility) of writes spreading across all system replicas, Federated essentially splits the difference between Raft and Eventual. However, Figure 5 shows that Federated fully replicates many more writes than Eventual, closely tracking the number of writes fully replicated by Raft.

The strong inner core of Raft replicas is the key to the Federated protocol tracking Raft's performance. Eventual replicas are biased in favor of performing anti-entropy with local replicas, allowing most anti-entropy sessions to perform quickly and without delay. By contrast, the Raft replicas in our Federated topology are intentionally spread across geographic regions. A new write originating at an Eventual replica is quickly spread to the local Raft replica, and is then broadcast to the rest of the regions via the Raft `AppendEntries` message. Disseminating writes quickly minimizes the possibility of another, later Eventual write starting up concurrently. Additionally, the Forte number prevents new forked writes from stomping on a conflicting write disseminated via Raft replicas.

Figure 7 shows the average number of stale reads, and the number of forked writes across different mean latencies. All three protocols perform similarly at smaller latencies, but Eventual and Federated deal with high latencies much more effectively than Raft.

Higher latencies affect Raft in at least two ways. First, higher latencies variability cause more out of order message. Second, we parameterize system timeouts by T which, in turn, is based on mean latencies. The result is that Raft's `AppendEntries` delay is longer for simulations with higher mean latencies, resulting in more conflicts. The same is true for anti-entropy delays, but the speed of Raft decisions is determined by the slowest quorum member, which can be quite slow when message variability is large. By contrast, a

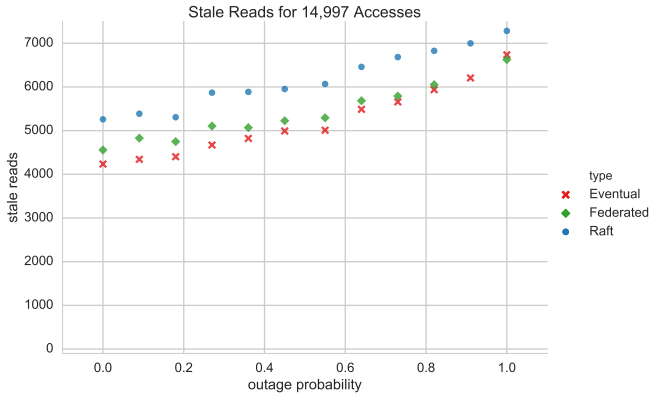

Fig. 3. Stale reads as outages increase.

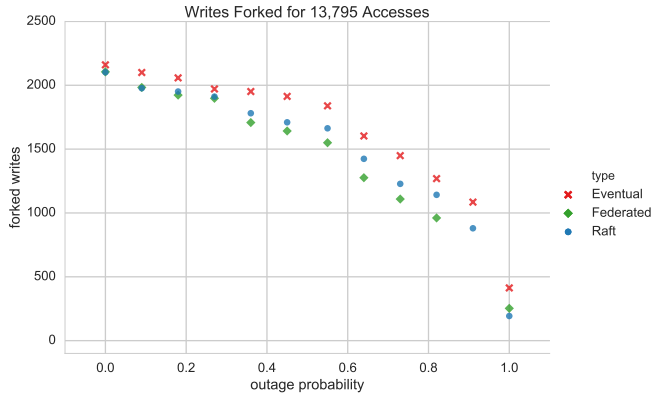

Fig. 4. Forked writes as outages increase.

slow anti-entropy participant only affects direct anti-entropy partners.

Figure 9 shows the mean cost of synchronous write operations to ongoing computations. Raft writes are not considered "done" until BLAH BLAH, whereas Eventual writes are considered done immediately, even before being replicated to other replicas. Most Federated replicas are Eventual, and Federated's average write cost therefore tracks Eventual's relatively closely.

VI. Discussion

A strong central core to provide support to the entire system has been suggested both in Oceanstore [27] and primary copy schemes [17]. We take this idea further in our experiments by Federating a strong central core composed of Replicas that perform consensus via the Raft consensus protocol and combine it with highly available eventually consistent systems. In this way Federated consistency gains the flexibility and availability of the Eventual replicas (leader re-election and no requirement for remote writes mean that the replica can continue even if it is completely partitioned from the rest of the network) while still getting guarantees from Raft, which minimizes "fork flipping" – the behavior of writing to one branch then another, truly pernicious inconsistent behavior that cannot be prevented in an eventually consistent system.

System designers can take advantage of heterogeneous replicas by implementing stronger consistency on more reliable machines that are able to handle more messages. Mobile replicas that are prone to network loss, out of order or missed messages, or other variable behavior can adapt their policy depending on the environment they're in. Our model also allows for *adaptive* behavior, in that the replicas can monitor the environment for change and as the mean latency decreases, adapt their T parameter accordingly. This is a no cost operation for Eventual replicas (who can also optimize pairwise gossip by implementing non-discrete random selection using Bandits or other optimization techniques), and only requires joint consensus on the part of the consensus group.

Our simulation implementation gave us the benefit of being able to manipulate many parameters to see how replication behavior changes. As a result, we only focused on the number of consistency messages sent. Potentially this overstates the differences between the protocols; Raft leaders have more processing requirements for example. Additionally bandwidth pressure and differences in messaging might be more important to the overall performance of a system. However, by focusing on the number of messages and isolating version replication from blob replication (e.g. minimizing the amount of data required for consistent behavior), we have allowed each protocol to fix some "message budget".

In truly resource constrained environments, allowing both Raft and Eventual (either in a homogenous or Federated context) to maintain an equal or semi-equal message budget by modifying the T parameter and allocating timing similarly means that congestion is reduced or optimized. Moreover, this system also allows for adaptive behavior by providing a single place of modification and optimization; T could be optimized according to a cost function (e.g. the message budget or other real time estimates of performance) or could be boosted for a replica that has to do a lot of work. This type of flexibility is important to being able to understand how minor changes in both the environment and protocols effect overall consistency in the system as a whole.

A. Fork Minimization

The primary form of inconsistency that we are concerned with is a fork, that is two separate, conflicting writes to the same parent version. In a homogenous eventually consistent system implemented with anti-entropy, the only way to minimize the number of forks is by increasing the speed of propagation. We have presented the best case for Eventual by implementing *bilateral* anti-entropy as well as neighbor selection likelihoods that lead to exponential convergence of a write. However, with a *latest-writer wins* policy, the possibility of "branch-flopping" exists; that is after a fork occurs, the latest version is updated on either side of the branch, and though the system will converge to a final write, branches may be arbitrarily long and represent significantly inconsistent states to the user.

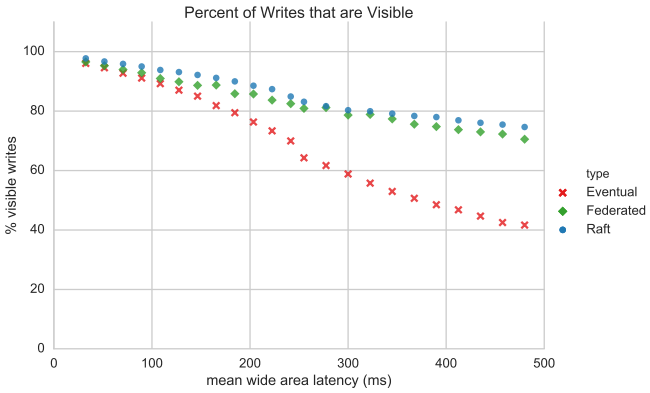

Fig. 5. The percentage of fully visible writes.

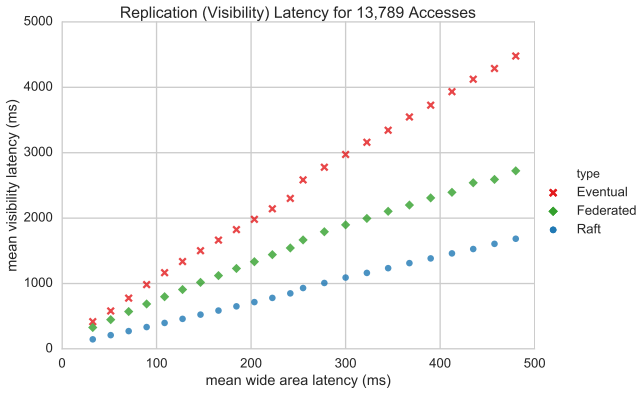

Fig. 6. The average full visibility latency.

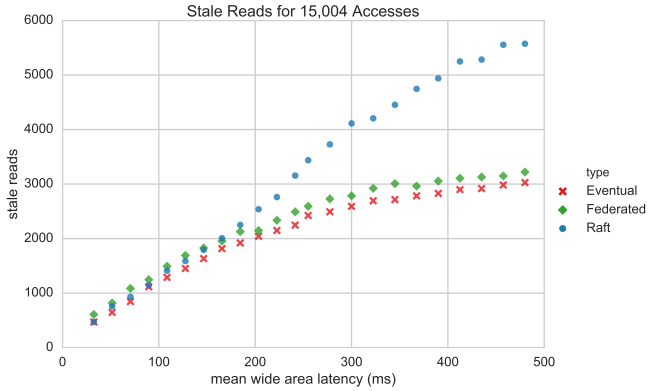

Fig. 7. The percent of reads that are stale in the system.

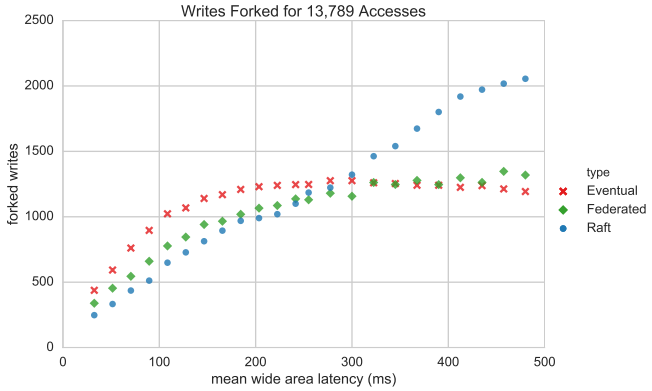

Fig. 8. The total number of conflicts (possible forks).

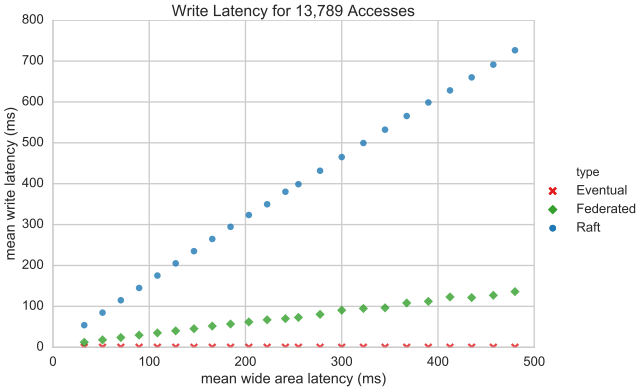

Fig. 9. Average local write cost.

A homogenous Raft system on the other hand simply does not allow forks to occur by requiring that the leader determines which write is accepted and which is not. Because we are aggregating writes in the `AppendEntries` messages every heartbeat interval, at most $n-1$ forks, where n is the number of replicas, are possible within the heartbeat interval. Whichever remote write arrives at the leader first will be accepted as the canonical branch. Here, the branch length is bounded to the number of writes possible by a single replica in the heartbeat interval, which is small – and the branch is eliminated at the heartbeat interval anyway.

The Federated system must at its core resolve how fork elimination is communicated to the Eventual system. A fork, by definition, is later than the earlier version. If the Federated system simply ignores or drops forks, the Eventual cloud might still propagate the fork around Raft. Worse, if the Eventual portion simply drops forks without deciding on a write to propagate, the system is no longer eventually consistent since it is possible that given no more writes half of the replicas make one fork visible while the other half make the other visible.

Note: right now we have Federated simply dropping the fork, which is creating bushier version trees and therefore more dropped writes.

VII. Related Work

One of the earliest attempts to hybridize weak and strong consistency was a model for parallel programming on shared memory systems by Agrawal et al [2]. This model allowed programmers to relax strong consistency in certain contexts with causal memory or pipelined random access in order to improve parallel performance of applications. Per-operation

consistency was extended to distributed storage by the Red-Blue consistency model of Li et al [31]. Here, replication operations are broken down into small, commutative sub-operations that are classified as red (must be executed in the same order on all replicas) or blue (execution order can vary from site to site), so long as the dependencies of each sub-operation are maintained. The consistency model is therefore global, specified by the red/blue ordering and can be adapted by redefining the ratio of red to blue operations, e.g. all blue operations is an eventually consistent system and all red is sequential.

The next level above per-operation consistency hybridization is called *consistency rationing* wherein individual objects or groups of objects have different consistency levels applied to them to create a global quality of service guarantee. Kraska et al. [25] initially proposed consistency rationing be on a per-transaction basis by classifying objects in three tiers: eventual, adaptable, and linearizable. Objects in the first and last groups were automatically assigned transaction semantics that maintained that level of consistency; however objects assigned the adaptable categorization had their consistency policies switched at runtime based on a cost function that either minimized time or write costs depending on user preference. This allowed consistency in the adaptable tier to be flexible and responsive to usage.

Chihoub et al. extended the idea of consistency rationing and proposed limiting the number of stale reads or the automatic minimization of some consistency cost metric by using reporting and consistency levels already established in existing databases [12], [13]. Here multiple consistency levels are being utilized, but only one consistency model is employed at any given time for all objects, relaxing or strengthening depending on observed costs. By utilizing all possible consistency semantics in the database, this model allows a greater spectrum of consistency guarantees that adapt at runtime.

Al-Ekram and Holt [4] propose a middleware based scheme to allow multiple consistency models in a single distributed storage system. They identify a similar range of consistency models, but use a middleware layer to forward client requests to an available replica that maintains consistency at the lowest required criteria by the client. However, although their work can be extended to deploying several consistency models in one system, they still expect a homogenous consistency model that can be swapped out on demand as client requirements change. Additionally their view of the ordering of updates of a system is from one versioned state to another and they apply their consistency reasoning to the divergence of a local replica's state version and the global version. Similar to SUNDR, proposed by Li et al. [32], an inconsistency is a fork in the global ordering of reads and writes (a "history fork"). Our consistency model instead considers object forks, a more granular level that allows concurrent access to different objects without conflict while still ensuring that no history forks can happen.

Hybridization and adaptation build upon previous work that strictly categorizes different consistency schemes. An alternative approach is to view consistency along a continuous scale with several axes that can be tuned precisely. Yu and Vahdat [45] propose the *conit*, a consistency unit described as a three dimensional vector that describes tolerable deviations from linearizability along staleness, order error, and numeric ordering. Similarly, Afek et al. [1] present quasi-linearizable histories which specify a bound on the relative movement of ordered items in a log which make it legally sequential.

VIII. Latency

Latency ranges for wide area networks can be extremely variable and are primarily determined by the last mile connection. While data centers often use backbone links to connect across the wide area, LTE networks, satellite networks, and the cable or fiber networks that connect users introduce extra round trip latency. Using ICMP to measure latency across the continental United States from the Princeton University network, Katz-Bassett et. al report local round trip latencies in the 10-50ms range and cross-country latencies in the 100ms range [22]. However, we believe that ICMP often gets preferential routing and that the simplicity of the echo protocol does not inform true message delays that would be experienced by a distributed system. We compared ICMP to a simple echo protocol implemented with GRPC [16] and discovered that the ICMP latency distribution is significantly smaller than GRPC in many environments.

IX. Conclusions

Our experiments utilized a very specific topology, and a natural question arises: what are the effects of different topologies on the performance of the system? Consider the case where we scale the number of eventually consistent replicas in each local area. More eventually consistent replicas would increase the anti-entropy propagation delay, increasing the likelihood of forks and stale reads and decreasing the number of writes that are fully replicated. The Raft core group could minimize the effect of this scaling to, however if the core Raft group remained fixed, it would also increase the load on the quorum for handling remote writes from the Eventual cloud and there would be increasingly more drops, particularly as forks are propagated around Raft followers. However, the increase in the size of the Eventual cloud would also give the system more fault tolerance as there are more paths to propagate updates when the Raft quorum is down, particularly across the wide area. Local outages may bring down Raft (partitioning the leader) but anti-entropy is resilient.

The failure mode that we considered took down wide area connections, if instead the failure mode was random replica failure, the system would respond differently. In a quorum size of 5, Raft can handle 2 failures before an extended outage. Leader failure would cause temporary outages until the election timeout occurs, but at least one append entries message will be missed. The central Raft group is therefore the most susceptible part of the system to random replica failure. If the Raft replica that fails is a follower, then the eventually

consistent replicas in that area can still make progress without a connection to the central quorum. Wide-area anti-entropy will allow updates to be propagated to the entire network without the central broadcast mechanism. However, the outage would probably lead to an increase in the number of forks as reads become increasingly stale with missed pairwise anti-entropy sessions reducing propagation speed.

In order to scale the topology and to increase the amount of failure tolerated, the central Raft quorum must also scale. However, increased quorum sizes often lead to decreased performance because the leader becomes a bottleneck as the number of messages increases. Additionally, the Raft quorum in our topology does not benefit from any localization like the eventually consistent replicas do. It is possible that placing all of the Raft replicas in a single location, giving Raft RPC messages the benefit of the local area connections and penalizing the synchronization of wide area Eventual replicas, would have increased the overall performance of the system. Because of the centrality of the Raft quorum in providing stronger consistency guarantees in a Federated environment, we propose future work into optimizing coordination for user-centric dynamic networks. In particular, we propose hierarchical consensus that will allow Raft to maintain smaller quorum sizes but scale to increasingly larger systems as well as localize decision making.

X. Conclusion

[PJK: Need to include a discussion of raft for every write].

In this paper we have presented a model for Federated consistency that allows individual replicas to expose local policies to users while still allowing for global guarantees given a heterogeneous system of replicas where at least a core group enforces sequential consistency. By designing a Federated system where only the interactions between replicas of varying consistency types are defined, systems can scale beyond the handful of devices usually described to dozens or hundreds of replicas in variable latency, partition-prone geographic networks. As each replica monitors its local environment and the throughput of its communication with other replicas it can adapt as necessary to the timeliness vs. correctness constraints required by the local user.

References

[1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[2] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Mixed consistency: A model for parallel programming. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 101–110. ACM, 1994.

[3] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[4] Raihan Al-Ekram and Ric Holt. Multi-consistency data replication. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 568–577. IEEE, 2010.

[5] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[6] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 International Conference on Management of Data*, pages 761–772. ACM, 2013.

[7] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[8] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014.

[9] David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 452–459. IEEE, 2011.

[10] David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer, 2013.

[11] U. Cetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin. Deno: A Decentralized, Peer-to-Peer Object Replication System for Mobile and Weakly-Connected Environments. *IEEE Transactions on Computers (TOC)*, 52(7), July 2003.

[12] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301. IEEE, 2012.

[13] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Consistency in the cloud: When money does matter! In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 352–359. IEEE, 2013.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[15] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the nineteenth Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, Bolton Landing, NY, USA, October 2003. ACM, ACM Press.

[16] Google. Google RPC. December 2016.

[17] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[18] Pat Helland. Immutability changes everything. *Queue*, 13(9):40, 2015.

[19] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[20] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[22] Ethan Katz-Bassett, John P. John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. Towards IP geolocation using delay and topology measurements. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 71–84. ACM, 2006.

[23] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.

[24] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[25] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[26] Sudha Krishnamurthy, William H. Sanders, and Michel Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 17–26. IEEE, 2002.

[27] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, and others. Oceanstore: An architecture for

global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[28] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[29] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[30] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[31] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[32] Jinyuan Li, Maxwell N. Krohn, David Mazieres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[33] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. *Proc. of the 11th USENIX NSDI*, 2014.

[34] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[35] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[36] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. Globalfs: A strongly consistent multi-site file system.

[37] Evaggelia Pitoura and Bharat Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Transactions on knowledge and data engineering*, 11(6):896–915, 1999.

[38] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[39] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase and Conference*, pages 391–430, 2000.

[40] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[41] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[42] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM*, 29, 1995.

[43] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Adrian Perrig, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[44] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[45] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.